A Fully Distributed Name Server for Reliable Distributed Applications

Ferris, Claudia S.
Cardozo, Eleri*
Instituto Tecnológico de Aeronáutica
Divisão de Engenharia Eletrônica
12228-900 São José dos Campos - SP
E-mail: ITA@BRFAPESP.BITNET

Abstract

In a loosely-coupled distributed system, resources are spread among hosts connected via network. If the system wants to appear as a single environment, it must offer a location mechanism to every resource in the network. One way of doing so is through the use of a name server.

This paper presents a distributed name server that offers three main services: a global number generator, a clock synchronization mechanism, that builds a global time base, and a location service, that does the mapping between resources and their locations. The proposed name server reaches a balance between reliability and availability by employing a scheme based on voting for the management of replicated data.

Every service provides ways to deal with host crashes and network partitions. Since the informations kept by the name server are distributed, a mechanism was adopted to synchronize updates and queries. Another point is the permanence of these informations that is obtained through a logging mechanism.

1 Introduction

The term distributed system has been applied to a wide range of multicomputer and multiprocessor systems. In this paper, we consider a distributed system to be "a group of multiple independent processors of different kinds, loosely-coupled, each one with its own

*DCA-FEE-UNICAMP, 13081-970 Campinas - SP.

private memory". Processing activities may be divided among processors and processors cooperate by exchanging messages over a network. The keyword here is *transparency*, meaning that the system appears to its users and applications as a centralized one: a virtual uniprocessor environment. So, it is important to notice that is the software, not the hardware, that characterizes a distributed system.

Most of the existent distributed systems follow the client/server model, where the client processes perform the system's tasks with the aid of server processes that answer requests from clients regarding accesses to resources[1], execution of operating system services, etc.

One of the aspects that must be considered in the implementation of a distributed system is *naming*. Each resource is identified by one or more symbolic names that map the resource's physical location. These names must be globally unique such that uttering a given name, anywhere in the system, always references the same resource. Naming is a key component of transparency. This implies that the users does not need to know the location of a resource to access it. Its a system's task to name resources in such a way that both local and remote resources are referred in the same way.

There are two relevant points related to naming:

- Transparency: The name of the resource has no relation with its location.
- Independency: The name of the resource does not have to be changed when the resource's location is altered.

A scheme based on independency is more adequate because it allows automatic migration of resources, but the vast majority of the existent systems implement a scheme based just on transparency.

The simplest way of implementing naming is to add the location of the resource to its name, but obviously this model has the disadvantage of not following both properties cited above.

Another possible scheme is *cache/broadcast* that consists in maintaining a little cache memory in each host to keep the locations of the last referred resources. Case a location is requested and is found out that it is not in the cache or it is out of date, a message is broadcasted in the network requiring the current location of the resource. Although this is a flexible method, it can cause overhead in the network.

Forward location pointers may be used to enhance most location schemes. The pointers serve as a reference to a new location of a resource. Each time a resource is moved from one host to another a pointer is left in its original place. To find a resource, it is necessary just to follow the chain of pointers. This method introduces an additional overhead to keep the pointers updated besides being liable to faults, such as pointers' lost due to processor crashes.

[1]Memory, peripheral devices, files, processors, programs, etc.

A fourth possible scheme is the *centralized name server*. The name server is formed from the registers of resources that want their locations be publicly known. So, to locate a resource the user sends a request with this name to the name server and the later replies with the location of the resource. This centralized scheme becomes undesirable when the system consists of many hosts that depend on just one name server and, in the case of a failure, all the system is compromised.

The next step is to distribute or replicate the name server among the hosts in the network, but not necessarily among all of them. There are two variations of this scheme.

In the first one, each name server maintains complete informations about the resources' location in such a way that all of them are capable of answering any location request.

In the second variation, the name servers have partial informations and cooperate among them to keep the informations consistent. When a name server receives a request that it can not handle, it forwards the request to another name server. There is also the possibility of each name server manage only unreplicated resources. To find a resource, a request must be broadcasted and one of the name servers will return the answer.

The name server proposed in this paper follows a distributed scheme, where all the name servers keep complete informations about the system's resources. Two important functions were added to the name server. A global number generator and a global time base. The design and implementation details of this distributed name server will be presented in the next sections.

2 Main Services in a Distributed System

2.1 Global numbering

In a distributed system, there can be many services being processed at the same time. As these services share the same environment, it is necessary to have a way to distinguish and order them, a sequential number that must be globally unique and generated in an ascendent order. These numbers, called global numbers, are used to identify services and maintain some event ordering. These numbers are useful, for example, in a logging operation where the ordering of events is important.

Thereby, a global number generator must be created with the task of assuring the properties above. The proposed algorithm is partially based on the algorithm by [Daniels88] that has a global number generator used to order log units. But different from Daniels' algorithm, this one uses a simpler scheme to assure that two simultaneous requests will be serialized.

The proposed name server is replicated among the hosts in the network and its communication with users' applications is done through an interface module. The name server has two ports to which the interface must connect itself to access the available services

that are offered through datagrams or stream connections. So, the user makes his request to the interface that has the task of accessing the right port and service in the name server. A scheme based on voting is employed for reliable global number generation.

The distributed algorithm for global number generation may be explained as follows: a user's application requests a global number to the interface. The interface receives the request and in order to guarantee that the number it will give is higher than those already generated, it will make a kind of voting to collect the greatest number previously generated by (N/2) + 1 hosts (all the name servers keep the last global number). Then, the interface asks for a connection with all the chosen hosts, which correspond to the participants of the voting process. Each name server answers with its global number and the interface chooses the greatest among them. The current global number is added one and all the elected hosts receive this new number. The name servers answer with acknowledgements signals and the interface returns the global number to the application.

As two requests for global numbers can not be processed concurrently, [2] the serializability of the process is guaranteed by the use of stream connections between the interface and the name servers. To make the process *atomic*, all the hosts that initiated in the voting process must stay until it finishes. In order to assure this, if during the voting process one of the name servers does not answer within a given time interval, the interface interrupts the voting process and interprets it as an error situation. In this case, the user requests again a global number until the request succeeds (it will succeed as soon as the host that failed to respond will be detected as down and removed from the next votings).

Another fact that must be considered in a network with replicated data is partition. This point will be better explained in the section 2.3. By now, it is sufficient to say that as the global number is kept by every name server, it is necessary to guarantee that if a partition occurs, there will be no chance of the same global number be generated at different sites, yielding an inconsistent situation. The algorithm bases the choice of participants in the voting process on the number of participants in the last voting, what will assure that every operation will contain the majority of existent updated copies and that only one partition will have the majority necessary. In order to distinguish a partition from a simple host disconnection, every time a name server disconnects, it performs a procedure that, among other functions, decreases the number of participants in the global number generation process.

2.2 Global time base

In many distributed systems the notion of time is of great importance. But, maintaining a global time base in an environment formed by multiple computers spatially separated (each one with its own clock) is not a simple task.

[2] In order to avoid the generation of repeated numbers.

The name server incorporates a synchronization method that results in a reliable global time base. It is an implementation of the algorithm proposed by [Vieira91]. This algorithm tries to be fault tolerant, have a low cost, being flexible and reliable. Different from other time schemes like the one presented by [Lamport78], it does not use timestamps to establish causal relations between two events. It exchanges messages and uses them to synchronize physical clocks. Even if the propagation delay of messages can not be exactly determined, it can be at least estimated, thus, all the messages will be guaranteed to be delivered and received within a maximum time interval *(dmax)*. The internal precision establishes that, at a given instant, two clocks will not drift more than a given time interval.

Suppose a network with N hosts (and N clocks). Let R^{int} be the time interval between resynchronizations, thus within every R^{int} time units, the clocks will be adjusted. This adjust will be based on *clock granularity*, that represents the clock resolution or precision, here called ng. So, a clock will be adjusted of 1 or 2 ng, according to some criteria. When a host's clock reaches the K^{th} period (that is, $K * R^{int}$), it broadcasts K, including to itself, in order to advise all the others about this event. The hosts wait for the messages from every other host and when a host counts $f + 1$ messages containing K, where f is the maximum number of faulty processors, it adjusts its clock with 1 or 2 ng, if it already reached K or not, respectively, and broadcasts a $SYNC$ message. Every host that receives this message synchronizes, too. Thereby, the first clock to synchronize signals the rest to be synchronized within *dmax* units of time. This algorithm is performed concurrently by all hosts in the network, maintaining them synchronized within a given precision. [Vieira91] presents formal proofs of the algorithm's accuracy. The algorithm was designed to support failures, if the number of active processes is at least twice the number of faulty ones (or $N > 2f$). Also, it avoids the backward effect, that is, a clock be adjusted backward.

2.3 Mapping

The third service offered by the name server is mapping which is, generally, the most common service offered by this class of system. The name server uses the informations it has to map an object to its location. As the proposed name server is distributed, also it is the information it maintains. This information is kept in data structures, like linked lists and hash tables.

A number of advantages can result from replicating information. Among this advantages are the increase in data accessibility, improved response time and reduction in network traffic, added to a system's throughput improvement by the ability of serving more than one request in parallel. These and other benefits must be balanced against the additional cost and complexities introduced by replication. One problem that arises is the synchronization of multiple accesses to copies of the same data structures.

Data copies may not always be kept identical because some of them may not be available at the time of an update or it is not desirable to apply an update to all of them, being enough to update a certain majority.

The operations that access the data structures must guarantee consistency as far as possible. This implies that a read operation must be based on a current copy and that a write operation can be applied to any copy of the same data structures[3]. Thereby, the name servers must be capable of determining which copies are current. This can be done, for example, through the use of version numbers. Only the hosts that are up to date have current version numbers.

The existing access control mechanisms can be characterized by the control disciplines they utilize. There is a class that uses a kind of centralized control and another that implements a decentralized control. This later presents more reliability and availability but, by the other side, it is also more complex.

A good example of a distributed mechanism of access control is *voting*. The idea is to assign some number of votes to read and write operations (read and write quorums) in such a way that the sum of these quorums is greater than the total number of potential voters. This ensures that there is a non-empty intersection between every read and write quorums, what guarantees that a read will always be based on a current version of a certain datum.

The read and write quorums may be chosen according to performance issues. Although the write quorum may be reduced to increase the performance of the operation, consistency must also be taken into account.

In a read operation, voting is necessary in order to assure that the data replied is the most recent one. In the write case, voting assures that a given number of hosts will be updated. Also, in the write operation, after each host performs updatings it has to send an acknowledgment to the host that began the process. Not only at this time, but along all the voting process, if one of the hosts does not answer, the process is stopped because it is assumed that an error occurred and the atomicity of the process was broken.

As said before, another fact that must be considered in a network with replicated data is partition. A partition occurs when the sites in the network split into disjoint groups that can not communicate with one another, but with members of the same group. Each one of this groups is called a partition. This can happen due to site or communication link failures. In the case of data being replicated, a dangerous situation can emerge when partitions occur: sites in one partition might perform updates in a datum while sites in another partition might perform different updates to a copy of the same datum. If this two updates conflict, it will be almost impossible to remedy this situation before communication between the involved partitions resumes. Thus, the mechanism of data

[3]If the copy is out of date, the missing updatings must be carried out before the requested updating begins.

control access has to choose between permitting updates in more than one partition, guaranteeing availability under the cost of inevitable conflicts, or permitting updates in at most one partition, in which case consistency is ensured at the price of reduced availability.

Algorithms in the first class are called optimistic because it is hoped that conflicts among updates are rare, and once they happen they could be detected and corrected. The second class of algorithms are called pessimistic because they consider consistency of greater importance than availability. The drawback with pessimistic algorithms is that as they permit update at just one distinguished partition, it is possible that failures occur in such a way that no updates can be performed anywhere in the system until these failures are repaired. Voting is the best known example of pessimistic algorithms.

The proposed name server tries to balance the need for availability and also the importance of consistency. The proposed voting scheme is based on pessimistic algorithms, specially, the algorithms by [Thomas79] and [Jajodia90] with some enhancements. Here follows an explanation about it.

As all the name servers deserve the same priority, everyone have the same vote weight that equals 1 (one), different from [Gifford79] that proposes a voting algorithm with different voting weights. During the initialization of the name server, it is established the quorum necessary to read and write operations. The write quorum will be, initially, a percentage of the active hosts at the instant of the voting and the read quorum will correspond to the remaining percentage in relation to the write quorum, added by one. This will guarantee a non-empty intersection between these two operations. As the response time in a name server is of great importance, the idea is to have a big write quorum and a little read quorum, thus, all the read requests can be attended reasonably fast and the write operation will "pay the price" of keeping a considerably number of copies equally consistent. Besides, as the choice for the quorums is made by the user, he has the chance of adapting them to his needs. To guarantee that only one partition will be authorized to make updates case a partition occurs, the read and write quorums will be *dynamic* because they will change as the number of hosts up changes. When this later changes, a partition may have occurred or a host may have disconnected itself from the system, case this number diminishes, or, by the other side, more hosts might have come up. Thus, these quorums will be recalculated for each data structure, either based on the number of participants in the last voting process that updated this data structure (case it has diminished), or based in the number of hosts up (case it has increased). Therefore, it is guaranteed that voting can only be performed if the partition contains the majority of the existent updated copies. As said before, this procedure, depending on how the partition occurs, may lead to a situation when no updates can be performed anywhere. To reduce the possibility of reaching this situation, the algorithm adopts the following procedure. If it is verified that the number of hosts up has diminished, the number of participants will be recalculated (quorum). Then, it is noticed that none of the partitions has enough

quorum to perform the voting. But it can be also verified that one of the partitions could perform it if it had just one more updated copy. In this case, it is permitted voting (and thus updatings) inside this single partition.

Initially, the number of participants will be equal to 1 (one) and, generally, this number will be at most equal to the number of active hosts. This number will be kept by every name server. Every time an update or a read is requested, the name server verifies if the number of participants remains less or equal to the active hosts. If it notices that the number of active hosts has changed, a new number of participants is calculated to see if there still have sufficient quorum to perform the requested operation. It is important to remember that it is possible to distinguish a partition from a voluntary disconnection, as explained in the section 2.1. Also, when a name server is initialized it starts with the most recent status of the existent data structures and variables, as well as the last global number generated.

The requests for voting are passed in a token among the hosts. When a host receives the token, it verifies if it is already participating in another voting process to the same data. The host votes either positive or negative and passes the token to the next host. This guarantees the serializability of the process. Each host verifies also if it is the one that completes the quorum. If the case, no more votes are needed and the token is returned to the host that began the operation.

For example, suppose that there are five active hosts in a network, A, B, C, D and E and that the original write and read participants number is 1 (Table 1). The write quorum was fixed in 70% and the read quorum in 30% + 1, what equals 4 and 2, respectively. Suppose also that there are no partitions in the network and that every site can communicate with each other.

Host	A	B	C	D	E
Write Quorum	1	1	1	1	1

Table 1: Initial situation

An update is requested to host B, and it verifies that it has the needed quorum to perform the operation. Then the number of participants will be made equal to 4 (Table 2).

Suppose now that a partition occurred and divided the network into two partitions A, B and C, D and E. Then, two update requests in the same data structure previously updated, are produced in both partitions, by C and D. The name server in C verifies that the number of active hosts (3) in its partition is now less than the last number of participants (4). This latter has to be recalculated and then equals 3. At first sight, this partition could not perform the update, but the difference between the number of hosts

Host	A	B	C	D	E
Write Quorum	1	4	4	4	4

Table 2: Hosts after the update has being done

up and the needed number of participants is one, the update is possible. First, the out of date host (A) will receive the missing version(s). Then, the update proceeds as in the case without partition. For the name server in D, the number of hosts up will be 2 and even when it recalculates the participants number it still does not have the required quorum, so this partition is forbidden to do updates. The same does not occur with reading, and, even with the partition, both could answer read requests, what guarantees the availability and also that the information given to the user is the most update it has in its partition.

Host	A	B	C	D	E
Write Quorum	3	3	3	4	4

Table 3: Hosts after the network partition and an update to one site

Afterwards, when the two sites join in one, the hosts that are not up to date can get updated copies, simply, by participating in a voting process.

A special situation occurs when the network splits into equal partitions. To solve this problem, the partition that has the host with the highest Internet address will be the one permitted to perform updatings.

If a host that has a non-updated copy receives a request for an update, another host that is asked to vote will return it the missing versions, and the operation proceeds. Also, if a host is requested to vote and the proposed version is more than one version greater than the one it has, it will not agree to vote and it will request the version(s) it has missed. All the messages are exchanged by datagrams and are identified by a header that corresponds to an operation of the voting process that is going on. There will be a log file where the last versions of the data structures is kept. This will be useful during the recovering of a host after a crash.

3 Persistence of The Name Servers' Informations

During a distributed system activity, all processing information is maintained in volatile memory (main memory) and if a crash occurs, this information will be completely lost. Due to the importance of the name server's task, it is necessary to preserve its informations

even in the case of failures. Thus, a mechanism must be used to make these informations permanent in such a way that the name server can be restarted in the same consistent state as it was at the time of the failure.

The protocol implemented is based on a *two-phase commit protocol*, [Chin91]. It will be executed together with the voting process that already implements atomicity. In the first phase of the protocol, a coordinator process sends messages of *pre-commit* to all the voting participants that, then, log the modifications in non volatile memory and reply with an acknowledgment to the coordinator. If one of then does not answer, the operation is aborted and the data is left in the same state it was before the updating started. Case the coordinator receives all the messages, a *commit* request is sent to them. Commit means that the name servers are allowed to make the modifications permanent, indicating the success of the updating process. Non destructive (read) operations do not need such protocol.

Considering the voting algorithm described in the previous sections, the first phase of the protocol will correspond to the moment that the host that began the operation (coordinator) sends the updatings to the voters. These ones update the logs in the disk and reply with acknowledgments. If any of them does not answer, the operation is aborted and everything remains as it was before the operation began. Otherwise, the coordinator sends *commit* requests to the voters which discards the old versions of the updated structures. This is the second phase of the protocol.

4 Conclusions

This paper presented a name server that tries to be very available and reliable. Differently from other name servers, it offers more than a resource location service. It implements a global time base and a global number generator.

The global number generator guarantees that no two requests are performed simultaneously. This is guaranteed with a very low overhead strategy employing just reliable (connected) communication.

Based on a synchronization method for real time distributed systems, the global time base offers a fault tolerant mechanism, under a low cost, that avoids the backward effect.

The voting process used to manage replication has the advantage of maintaining consistency even if a partition occurs, without penalties to queries. Serializability among updatings are obtained by employing a token-passing scheme efficiently implemented with datagrams.

The three services handle situations as total crashes or network partitions, always applying the atomicity property. Every relevant information is made permanent in a logging scheme. This permits the recoverability of the system after a crash.

This name server implements a vast number of properties desired in a distributed

system. It has been implemented and tested in a local network formed by SUN, HP and DEC workstation. The messages are serialized using eXternal Data Representation (XDR) protocol, assuring compatibility among different platforms. The name server is been implemented in *C* with about 60% of its code already completed.

References

[Daniels88] Daniels, D.S.
Distributed Logging for Transaction Processing, Phd Thesis, Carnegie Mellon University, 1988.

[Lamport78] Lamport, L.
Time, Clocks and The Ordering of Events in a Distributed System, Communications of the ACM, 21(7), July, 1978.

[Thomas79] Thomas, R.H.
A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases, ACM Transactions on Database Systems, 4(2), June, 1979.

[Jajodia90] Jajodia, S.; Mutchler, D.
Dynamic Voting Algorithms for Maintaining The Consistency of a Replicated Database, ACM Transactions on Database Systems, 15(2), June, 1990.

[Gifford79] Gifford, D.K.
Weighted Voting for Replicated Data, ACM Proceedings of the 7th Symposium on Operating Systems Principles, December, 1979.

[Chin91] Chin, R.S.; Chanson, S.T.
Distributed Object-Based Programming Systems, ACM Computing Surveys, 23(1), March, 1991.

[Vieira91] Vieira Jr., L.; Fraga, J.S.,
Estabelecimento de uma Base de Tempo Global em Sistemas Distribuídos em Tempo Real Utilizando um Algoritmo de Sincronização de Relógios Físicos, 8 o CBA, Belém, PA, 1990.